



Deep Copy and Unified Memory in OpenMP

Denver, CO ♦ August, 2017



Why both?

Or: If everyone has unified memory why map?

- I. Mapping provides more information to both the compiler and the runtime
- 2. Not all hardware has unified memory
- 3. Not all unified memory is the same
 - I. Can all memory be accessed with the same performance from everywhere?
 - 2. Do atomics work across the full system?
 - 3. Are flushes required for coherence? How expensive are they?

Specifying unified memory in OpenMP

- OpenMP does not require unified memory
 - Or even a unified address space
- This is not going to change



How do you make non-portable features portable?

- Specify what they provide when they are present
- Give the user a way to assert that they are required
- Give implementers a way to react to that assertion



One solution: Extension declarations

```
#pragma omp extension [extension clauses...]
```

Extension name	Effect
unified_address	Guarantee that device pointers are unique across all devices, is_device_ptr is not required
unified_memory	Host pointers are valid device pointers and considered present by all implicit maps, implies unified_address, memory is synchronized at target task sync
system_atomics	OpenMP atomic operations work across devices, may require extra flags
system_coherence	Combined with unified_memory, explicit flushes and synchronization are not required, even across devices



OpenMP Extension example #1

```
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```



OpenMP Extension example #1

```
int * arr = even...  
#pragma omp ... at teams distribute parallel for  
for (int i=0; i<1000; i++){  
    arr[i] = i  
}
```



OpenMP Extension example #1

```
#pragma omp extension unified_memory
int * arr = new int[50];
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```



OpenMP Extension example #1

```
#pragma omp extension aligned_memory
int * arr = new int[1000];
#pragma omp target
for (int i=0; i<1000;
    arr[i] = i;
}
```



OpenMP Extension example #2

```
int * arr = omp_target_alloc(sizeof(int)*50,
                           omp_get_default_device());
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```



OpenMP Extension example #2

```
int * arr = mp...omp_get_device());  
#pragma omp ai...omp_get_default_device());  
for (int i=0 i...omp_get_default_device());  
    arr[i] = i  
}
```



OpenMP Extension example #2

```
#pragma omp extension unified_address
int * arr = omp_target_alloc(sizeof(int)*50,
                               omp_get_default_device());
#pragma omp target teams distribute parallel for
for (int i=0; i<50; ++i){
    arr[i] = i;
}
```



OpenMP Extension example #2

```
#pragma omp extension
int * arr = omp_targeted_address(sizeof(int)*50,
                                    default_device());
#pragma omp target
for (int i=0; i<50; i++)
    arr[i] = i;
}
```



Deep copy today

- It **is** possible to use deep copy in OpenMP today
- Manual deep copy works by *pointer attachment*



Pointer attachment

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
myvec_t v = init_myvec();  
#pragma omp target map(v, v.data[:v.len])  
{  
    do_something_with_v(&v);  
}
```



Pointer attachment

Map structure v

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
myvec_t v = init_myvec();  
#pragma omp target map(v, v.data[:v.len])  
{  
    do_something_with_v(&v);  
}
```



Pointer attachment

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
myvec_t v = init_myvec();  
#pragma omp target map(v, v.data[:v.len])  
{  
    do_something_with_v(&v);  
}
```

Map structure v

Map data array and attach to v

What's the downside?

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
size_t num = 50;  
myvec_t *v = alloc_array_of_myvec(num);  
#pragma omp target map(v[:50], ??????)  
{  
    do_something_with_v(&v);  
}
```



What's the downside?

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
size_t num = 50;  
myvec_t *v = alloc_array_of_myvec(num);  
#pragma omp target map(v[:50], ??????)  
{  
    do_something_with_v(&v);  
}
```

Now we need a loop,
more breaking up the
code!



The future of deep copy: User-defined mappers

```
#pragma omp declare mapper(<type> <var>)
  [name(<name>)]
  [use_by_default]
  [map(<list-items>...)...]
```

- Allow users to define mappers in terms of normal map clauses
- Offer extension mechanisms to pack or unpack data that can't be bitwise copied, or expressed as flat maps



Our array example

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
#pragma omp declare mapper(myvec_t v)\n        use_by_default map(v, v.data[:v.len])  
size_t num = 50;  
myvec_t *v = alloc_array_of_myvec(num);  
#pragma omp target map(v[:50])  
{  
    do_something_with_v(&v);  
}
```



Our array example

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
  
#pragma omp declare mapper(myvec_t v)\n        use_by_default map(v, v.data[:v.len])  
size_t num = 50;  
myvec_t *v = alloc_array_of_myvec(num);  
#pragma omp target map(v[:50])  
{  
    do_something_with_v(&v);  
}
```

No loop required, no extra code at usage, just map

Composition of mappers

```
typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;
#pragma omp declare mapper(myvec_t v) \
    use_by_default map(v, v.data[:v.len])
typedef struct mypoints {
    struct myvec * x;
    struct myvec scratch;
    double useless_data[500000];
} mypoints_t;

#pragma omp declare mapper(mypoints_t p) \
    use_by_default \
    map(/* self only partially mapped, useless_data can be ignored */ \
        p.x, p.x[:1]) /* map and attach x */ \
    map(alloc:p.scratch) /* never update scratch, including its internal maps */

mypoints_t p = new_mypoints_t();
#pragma omp target
{
    do_something_with_p(&v);
}
```



Composition of mappers

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
#pragma omp declare mapper(myvec_t v)\  
    use_by_default map(v, v.data[:v.len])  
typedef struct mypoints {  
    struct myvec * x;  
    struct myvec scratch;  
    double useless_data[500000];  
} mypoints_t;  
#pragma omp declare mapper(mypoints_t p) \  
    use_by_default \  
    map(/* self only partially mapped, useless_data can be ignored */\  
        p.x, p.x[:1]) /* map and attach x */\ \  
    map(alloc:p.scratch) /* never update scratch, including its internal maps */\  
  
mypoints_t p = new_mypoints_t();  
#pragma omp target  
{  
    do_something_with_p(&v);  
}
```

Pick and choose what to map



Composition of mappers

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
#pragma omp declare mapper( myvec_t ) use_by_default map(  
    myvec_t *x, double *y);  
  
typedef struct mypoints {  
    struct myvec *x;  
    struct myvec scratch;  
    double useless_data[100000];  
} mypoints_t;  
  
#pragma omp declare mapper(mypoints_t p) \  
    use_by_default \  
    map(/* self only partially mapped, useless_data can be ignored */\  
        p.x, p.x[:1]) /* map and attach x */ \  
    map(alloc:p.scratch) /* never update scratch, including its internal maps */  
  
mypoints_t p = new_mypoints_t();  
#pragma omp target  
{  
    do_something_with_p(&p);  
}
```

Re-use the myvec_t mapper

Pick and choose what to map



Composition of mappers

```
typedef struct myvec {  
    size_t len;  
    double *data;  
} myvec_t;  
#pragma omp declare mapper( myvec_t ) use_by_default map(  
    myvec_t *x, double *y);  
  
typedef struct mypoints {  
    struct myvec *x;  
    struct myvec scratch;  
    double useless_data[100000];  
} mypoints_t;  
  
#pragma omp declare mapper(mypoints_t p) \  
    use_by_default \  
    map(/* self only partially mapped, useless_data can be ignored */\  
        p.x, p.x[:1]) /* map and attach x */ \  
    map(alloc:p.scratch) /* never update scratch, including its internal maps */  
  
mypoints_t p = new_mypoints_t();  
#pragma omp target do_something_with_p(&v);  
{  
    do_something_with_p(&v);  
}
```

Re-use the myvec_t mapper

Pick and choose what to map

No explicit map required!



Defining mappers from explicit serialization and deserialization

- Declare mappers by stages, all are replaceable
 - alloc
 - pack_to
 - unpack_to
 - pack_from
 - unpack_from
 - release
- Any arbitrary data can be mapped



Conclusions

- OpenMP **needs** more refined control over memory visibility
- Unified memory is becoming common, but not all unified memory is created equal
- Deep copy with composition can make mapping less painful

